```python
import numpy as np
import sys

class RNeuralNetwork:
    def __init__(self, input_size=1, hidden_neurons=10, output_size=1):
        np.random.seed(1)
        self.input_size = input_size
        self.hidden_neurons = hidden_neurons
        self.output_size = output_size

        # Weight matrix connecting input to hidden layer
        self.w1 = np.random.randn(self.input_size, self.hidden_neurons) * 0.01
        # Bias vector for the hidden layer
        self.b1 = np.zeros((1, self.hidden_neurons))
        # Weight matrix connecting hidden layer to output layer
        self.w2 = np.random.randn(self.hidden_neurons, self.output_size) * 0.01
        # Bias vector for the output layer
        self.b2 = np.zeros((1, self.output_size))

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def sigmoid_derivative(self, z):
        return self.sigmoid(z) * (1 - self.sigmoid(z))

    def forward(self, X):
        # Weighted input to the hidden layer (before activation)
        self.z1 = np.dot(X, self.w1) + self.b1
        # Activation of the hidden layer (output of sigmoid applied to z1)
        self.a1 = self.sigmoid(self.z1)
        # Weighted input to the output layer (before activation)
        self.z2 = np.dot(self.a1, self.w2) + self.b2
        return self.z2

    def compute_loss(self, y_pred, y_true):
        return np.mean((y_pred - y_true) ** 2)

    def backward(self, X, y, y_pred):
        n_samples = X.shape[0]

        # Gradient of loss with respect to z2 (output layer pre-activation)
        dL_dz2 = 2 * (y_pred - y) / n_samples
        # Gradient of loss with respect to w2 (weights between hidden and output layers)
        dL_dw2 = np.dot(self.a1.T, dL_dz2)
        # Gradient of loss with respect to b2 (biases of the output layer)
        dL_db2 = np.sum(dL_dz2, axis=0, keepdims=True)

        # Gradient of loss with respect to a1 (hidden layer activations)
        dL_da1 = np.dot(dL_dz2, self.w2.T)
        # Gradient of loss with respect to z1 (hidden layer pre-activation)
        dL_dz1 = dL_da1 * self.sigmoid_derivative(self.z1)
        # Gradient of loss with respect to w1 (weights between input and hidden layers)
        dL_dw1 = np.dot(X.T, dL_dz1)
        # Gradient of loss with respect to b1 (biases of the hidden layer)
        dL_db1 = np.sum(dL_dz1, axis=0, keepdims=True)

        return dL_dw1, dL_db1, dL_dw2, dL_db2

    def update_parameters(self, gradients, learning_rate):
        dL_dw1, dL_db1, dL_dw2, dL_db2 = gradients

        self.w1 -= learning_rate * dL_dw1
        self.b1 -= learning_rate * dL_db1
        self.w2 -= learning_rate * dL_dw2
        self.b2 -= learning_rate * dL_db2
```

```
    def train(self, X, Y, max_epochs=5000, learning_rate=0.01, improvement_threshold=1e-6¬
):
        prev_loss = float('inf')
        losses = []  # Track losses for visualization
        for epoch in range(max_epochs):
            y_pred = self.forward(X)
            loss = self.compute_loss(y_pred, Y)
            losses.append(loss)

            # Print diagnostics every 10000 epochs
            if (epoch + 1) % 10000 == 0:
                print(f"Epoch {epoch + 1}/{max_epochs}, Loss: {loss:.6f}")
                print(f"w1 mean: {np.mean(self.w1):.5f}, std: {np.std(self.w1):.5f}")
                print(f"b1 mean: {np.mean(self.b1):.5f}, std: {np.std(self.b1):.5f}")
                print(f"w2 mean: {np.mean(self.w2):.5f}, std: {np.std(self.w2):.5f}")
                print(f"b2 mean: {np.mean(self.b2):.5f}, std: {np.std(self.b2):.5f}")
                print(f"Gradient Norm: {np.linalg.norm(self.w1) + np.linalg.norm(self.w2)¬
:.5f}")

            gradients = self.backward(X, Y, y_pred)
            self.update_parameters(gradients, learning_rate)

            if 0:
                if prev_loss - loss < improvement_threshold and epoch > 500:
                    print(f"Stopping early at epoch {epoch + 1}, Loss: {loss:.6f}")
                    break

            prev_loss = loss

    def predict_formula(self):
        # Rough approximation to determine function type
        if np.allclose(self.w1, 0):
            return "Linear"
        elif np.any(self.w1 > 0) and np.any(self.w1 < 0):
            return "Nonlinear"
        else:
            return "Cannot determine"

if   name   == "  main  ":
    def load_data(filename):
        X_list, Y_list = [], []
        with open(filename, 'r') as f:
            for line in f:
                if line.strip():
                    x, y = map(float, line.split())
                    X_list.append(x)
                    Y_list.append(y)

        X = np.array(X_list).reshape(-1, 1)
        Y = np.array(Y_list).reshape(-1, 1)

        # Normalize to [0, 1] range
        X_min, X_max = X.min(), X.max()
        Y_min, Y_max = Y.min(), Y.max()
        X = (X - X_min) / (X_max - X_min)
        Y = (Y - Y_min) / (Y_max - Y_min)

        return X, Y, X min, X max, Y min, Y max

    if len(sys.argv) != 2:
        print("Usage: python script.py <data_file>")
        sys.exit(1)

    data_file = sys.argv[1]
```

```
    print(f"data_file = {data_file}")
    X, Y, X_min, X_max, Y_min, Y_max = load_data(data_file)
    print("First few samples (original and normalized):")
    for i in range(min(5, len(X))):
        x_original = X[i][0] * (X_max - X_min) + X_min
        y_original = Y[i][0] * (Y_max - Y_min) + Y_min
        print(f"x(normalized)={X[i][0]:.5f}, y(normalized)={Y[i][0]:.5f} | x(original)={x¬
_original:.5f}, y(original)={y_original:.5f}")

    # Initialize and train the model
    nn = RNeuralNetwork(input_size=1, hidden_neurons=10, output_size=1)
    nn.train(X, Y, max_epochs=300000, learning_rate=0.01)

    # Show prediction formula (approximation)
    formula_type = nn.predict_formula()
    print(f"The network predicts the relationship type as: {formula_type}")

    print("First few predictions (for verification):")
    for i in range(min(5, len(X))):
        y_pred = nn.forward(X[i].reshape(1, -1)) * (Y_max - Y_min) + Y_min
        x_original = X[i][0] * (X_max - X_min) + X_min
        print(f"x={x_original:.5f}, Predicted y={y_pred[0][0]:.5f}")

    # Test predictions
    while True:
        user_input = input("Enter x (original scale) to predict y (or 'quit' to exit): ")
        if user_input.lower() == 'quit':
            break
        try:
            x_new = float(user_input)
            x_new_norm = (x_new - X_min) / (X_max - X_min)
            x_new_array = np.array([[x_new_norm]])
            y_pred_norm = nn.forward(x_new_array)
            y_pred = y_pred_norm * (Y_max - Y_min) + Y_min
            print(f"Predicted y for x={x_new}: {y_pred[0][0]:.5f}")
        except ValueError:
            print("Invalid input. Please enter a numeric value.")
```